Two world views

FP: functions that perform some operation
OOP: classes that give behavior to some kind of data

Which is better? Depends on software evolution, taste.
Can awkwardly emulate each other

Common pattern: expressions

Operations over type of data

|       | eval | toString | usesX | ...
|-------|------|----------|-------|-----
| VarX  |      |          |       |     |
| Sine   |      |          |       |     |
| Times  |      |          |       |     |
| ...    |      |          |       |     |

Variants of a type of data

FP: behavior by operation

Function per operation with branch per variant

Datatype with constructor per variant

Pattern-matching selects variant.
Wildcard can merge rows in a function.

OOP: behavior by variant

Base class with (abstract) method per operation

Subclass per variant overrides each operation method to implement variant's behavior

Dynamic dispatch selects variant.
Concrete method in base class can merge rows where not overridden.
FP: Extensibility

<table>
<thead>
<tr>
<th>eval</th>
<th>toString</th>
<th>usesX</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Times</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sqrt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add variant:
add constructor,
change all functions over datatype

Add operation:
add function,
no other changes

ML type-checker gives “to-do list”
via inexhaustive pattern-match warnings

OOP: Extensibility

<table>
<thead>
<tr>
<th>eval</th>
<th>toString</th>
<th>usesX</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Times</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sqrt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add variant:
add subclass,
no other changes

Add operation:
add method
to base class and all subclasses

Java/Scala type-checker gives “to-do list”
via errors about non-overidden
abstract method

Thoughts on Extensibility

Making software extensible is valuable and hard.
• If new operations likely, use FP
• If new variants likely, use OOP
• If both, use somewhat odd “design patterns”
• Reality: The future is hard to predict!

Extensibility is a double-edged sword.
• Code more reusable without being changed later
• Original code more difficult to reason about locally or change later
  without breaking remote extensions
• Language mechanisms also support restricting extensibility:
  • ML abstract types
  • Java’s final prevents subclassing/overriding

Binary Operations

What about operations that take two arguments of possibly different variants?
• Include value variants Int, Rational, ...
• (Re)define Add to work on any pair of Int, Rational, ...
The addition operation alone is now a different 2D grid:
ML approach: pattern-matching

Natural: pattern-match both simultaneously

fun add_values(v1,v2) =
  case (v1,v2) of
    (Int i, Int j) => Int (i+j)
  | (Int i, Rational(j,k)) => Rational (i*k+j, k)
  | (Rational _, Int _) => add_values (v2,v1)
  ...

fun eval e =
  case e of
    ...
    | Add(e1,e2) => add_values (eval e1, eval e2)

OOP approach: dynamic dispatch

abstract class Value extends Expr {
  def addValues(v: Value): Value
}
class Add extends Expr {
  ...
  override def eval(): Value = {
    e1.eval().addValues(e2.eval())
  }
}
class MyInt extends Value {
  // add this to v
  override def addValues(v: Value): Value = ...
  // what goes here?
}

Reflecting

• Double dispatch manually emulates basic pattern-matching.
• Does it change the way in which OOP handles evolution?
• If we add an operation over pairs of Values:
  • OOP double dispatch: how many classes are added? How many change?
  • FP pattern matching: how many functions are added? How many change?
• If we add a kind of Value:
  • OOP double dispatch: how many classes are added? How many change?
  • FP pattern matching: how many functions are added? How many change?
• What if we could dispatch based on all arguments at once?
Multimethods

General idea:
• Allow multiple methods with same name and # arguments
• Indicate which ones take instances of which classes
• Use dynamic dispatch on all arguments in addition to receiver to pick which method is called
• NOT same as static overloading.

If dynamic dispatch is essence of OOP, this is cleaner, more OOP

Downside:
  subclassing sometimes causes “no clear winner” for which method to call

Research idea picked up in some recent languages (e.g., Clojure, Julia)

The other way is possible with planning.

• Functions allow new operations and objects allow new variants without modifying existing code even if they didn’t plan for it.

• Functions can support new variants “if they plan ahead”
  • Use type constructors to make datatypes extensible
  • Operations use function argument to give result for extension

• Objects can support new operations “if they plan ahead”
  • Visitor Pattern uses double dispatch to allow new operations “on the side”
  • See assignment.

• Neither “plan ahead” option is elegant, but they work.

Closures vs. Objects

Closure:
• Captures code of function, by function definition.
• Captures all bindings the code may use, by lexical scope of definition.

Object:
• Captures code for all methods that could be called on it, by class hierarchy.
• Captures bindings that may be used by that code, by instance variables declared in class hierarchy.

Emulation in both directions is fascinating.